Loyalty-based Selection: Retrieving Objects That Persistently Satisfy Criteria

Zhitao Shen, Muhammad Aamir Cheema, Xuemin Lin
The University of New South Wales, Australia
{shenz,macheema,lxue}@cse.unsw.edu.au

ABSTRACT

A traditional query returns a set of objects that satisfy user defined criteria at the time query was issued. The results are based on the values of objects at query time and may be affected by outliers. Intuitively, an object better meets the user's needs if it persistently satisfies the criteria, i.e., it satisfies the criteria for majority of the time in the past T time units. In this paper, we propose a measure named *loyalty* that reflects how persistently an object satisfies the criteria. Formally, the loyalty of an object is the total time (in past T time units) it satisfies the query criteria. In this paper, we study top-k loyalty queries over sliding windows that continuously report k objects with the highest loyalties. Each object issues an update when it starts satisfying the criteria or when it stops satisfying the criteria. We show that the lower bound cost of updating the results of a top-k loyalty query is $O(\log N)$, for each object update, where N is the number of updates issued in last T time units. We conduct a detailed complexity analysis and show that our proposed algorithm is optimal. Moreover, effective pruning techniques are proposed to improve the efficiency. We experimentally verify the effectiveness of the proposed approach by comparing it with a classic sweep line algorithm.

Categories and Subject Descriptors

H.2 [**Database Management**]: Miscellaneous

General Terms

Algorithms,Performance,Experimentation

Keywords

Data Streams, Loyalty Queries, Temporal Data

1. INTRODUCTION

A traditional query Q returns every object that satisfies the query criteria at the time t query was issued. The traditional queries do not consider the history of the objects' values, i.e., the values of objects in the recent past. Hence, the traditional queries fail to capture how persistently an object satisfies the query criteria. Consider the example of a stock broker who issues a query at time t to retrieve the profitable stocks. He may define a set of criterions to denote the profitability. A traditional query returns every stock s that satisfies the criterions at time t. Although a returned stock s meets the criteria at time t, the history of the stock s may indicate that it usually does not satisfy the criteria and is not a good choice for investment. Hence, a query that does not take into account the history of stock items is not suitable.

To address the above mentioned problem, in this paper, we propose a new query operator called *loyalty queries*. A loyalty query considers how persistently the objects satisfy the query criteria. Consider a *traditional query* Q that defines a set of criterions. Let $Q(o,t)$ denote whether an object o satisfies the criteria of query Q at time t or not. More specifically, $Q(o,t)$ is true if and only if the object o satisfies the query criteria at time t. Let T be a user defined parameter. The loyalty of an object o is the total time duration for which $Q(o,t)$ is true within last T time units. The measure is called "loyalty" because it signifies how persistently the object o meets the criteria in the recent past. In this paper, we study continuous *top-k* loyalty queries that continuously report k objects with the highest loyalties.

Loyalty queries have many interesting applications in different areas such as location based services, wireless sensor network, stock market, traffic monitoring, and internet applications, etc. For instance, in the example of the stocks, the stock broker may retrieve top-k loyal objects to retrieve better options for investment. Consider another example of a paid parking system that notifies the nearby cars of its availability, i.e., the cars that are in its *influence zone* [4] or the cars that are within 1 Km of the parking space [3]. At a given time t, the system may send SMS to some cars that satisfy the criteria (e.g. a car that lies within 1Km at time t). However, most of such cars may just be passing through that area and may not be interested in parking. On the other hand, a car that satisfies the criteria for majority of the time in recent past may actually be looking for the parking. Hence, the system may use top-k loyalty queries to send notifications to such cars.

We next summarize our contributions in this paper.

Novel query operator. To the best of our knowledge, we are the first to study continuous loyalty queries. In this paper, we formalize the definition of loyalty queries and present a framework that efficiently solves the loyalty queries.

Continuous updates. We study the problem in a continuous time domain where the updated results are reported as soon as the results change as opposed to the *time-stamp* model where the results are updated after every u time units. Note that the time-stamp model suffers from either high computational cost or low accuracy. More specifically, if u is small, the computation cost increases because the results are to be updated more often. On the other hand, if u is large, the accuracy is reduced because the results may have become invalid between two successive time-stamps. The continuous updates provided by our algorithm do not have these limitations.

Optimal computation cost. An object issues an update if it starts satisfying the query criteria or if it stops satisfying the query cri-

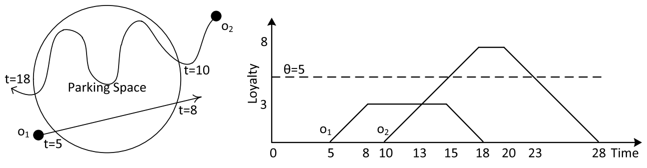

Figure 1: Example of Loyalty Queries

teria. Note that the top-k loyal objects may change whenever an object issues an update. Let N be the total number of object updates issued in the last T time units. Upon receiving an object update, our algorithm updates the top-k loyal objects in $O(\log N)$. We prove that this is the lower bound update cost for top-k loyalty queries, hence our algorithm is optimal.

Extensive evaluation and analysis. We theoretically analyse the complexity of our algorithm and prove that it meets the lower bound cost. We also conduct experiments to show the effectiveness and the efficiency of our proposed approach. We compare our algorithm with the Bentley-Ottmann sweep line algorithm [2]. For N object updates, the total cost of the Bentley-Ottmann algorithm is $O(N^2 \log N)$ in the worst case. In contrast, the total worst case cost of our algorithm is $O(N \log N)$. Extensive experiments conducted on both real and synthetic data sets demonstrate that our proposed approach is an order of magnitude faster than the Bentley-Ottmann algorithm.

Due to the space limits, we do not include all the technical details and experimental results in the paper. See our technical report [9] for the further details.

2. PROBLEM DEFINITION

Traditional queries. A traditional query Q defines a set of criteria. Given an object o and a timestamp t, we use $Q(o, t)$ to denote whether o satisfies the query criteria of Q at t. For ease of presentation we define $Q(o, t)$ using a step function.

$$Q(o,t) = \begin{cases} 1 & \text{if } o \text{ satisfies the query critera of } Q \text{ at } t; \\ 0 & \text{if } o \text{ does not satisfy the query critera of } Q \text{ at } t. \end{cases}$$

Consider an application for monitoring the cars around the parking space and the parking system notifies the cars with high loyalties in Figure 1. Given two moving objects(cars) o_1 and o_2, o_1 enters the space at time 5 and leaves at time 8. o_2 enters at time 10 and leaves at time 18. Therefore, $Q(o_1, 6) = 1$ and $Q(o_2, 6) = 0$.

Sliding windows. Usually users are not interested in the entire past history of the data stream but rather the recent data over sliding windows. In this paper, we consider a data stream model in the continuous time domain. For a fixed length of time period T, a sliding window contains all the objects and the corresponding attributes within last T time units.

Loyalty of an object. Given a traditional query Q and a sliding window size T, we define the loyalty of an object o at time t, $loyalty(o, t) = \int_{t-T}^{t} Q(o, x)dx$

The loyalty of an object o shows how long o is a query result of Q during the last T time units. Without loss of generality, in this paper we prefer the objects with higher loyalties.

Consider a sliding window of size 10 ($T = 10$) in Figure 1. Then, $loyalty(o_1, 8)$ (the loyalty of o_1 at time 8) is 3 because o has been around the parking space for 3 time units. Note that the coordinates in Figure 1 present the loyalties of o_1 and o_2 as the time t changes. Similarly, we can see that $loyalty(o_2, 13) = 3$.

Top-k loyalty queries. Consider a set of objects O, a traditional query Q, a sliding window size T and a parameter k. The top-k loyalty query at time t returns an answer set from O that consists of k objects such that for every object o in the answer set and for any other $o' \in O$, $loyalty(o, t) \geq loyalty(o', t)$.

Consider the example in Figure 1. If we monitor the top-1 loyal object and the window size T is 10, o_1 is the result of the top-1 loyalty query from 5 to 13 and o_2 is the result from 13 to 28.

Continuous queries. In this paper, we study the continuous loyalty queries, namely, we issue the query once and it monitors the query results continuously. Since we solve the queries in the continuous time domain, it is impossible to compute the results for an infinite number of time snapshots. In this paper we shows that although the loyalty of an object is changing over time, we do not need to update the loyalty and the query results for every time snapshot.

3. FRAMEWORK

In real world scenarios, given a set of objects of observation O, the objects may be distributed and users may want to know the global results of a loyalty query. Thus, we present a general framework that aims to handle the loyalty queries in both various environments. Our framework consists of two main components: the traditional query module and the loyalty query module.

Traditional query module. Given a traditional query Q (e.g., a range query), each object issues an *update* when it starts satisfying the query criteria or when it stops satisfying the criteria. More specifically, traditional query module reports whenever the value of $Q(o, t)$ is changed for any object o.

Object updates. Given a query Q and an object o, we say there is an update u of o at time t if the derivative of Q at t is infinity, i.e., $\frac{d}{dt}Q(o, t) = \infty$. In other words, $Q(o, t)$ changes at time t.

In figure 1, a moving object issues the update only when it enters or exits the monitoring space. Therefore, o_1 reports two updates at time 5 and 8, and o_2 reports two updates at time 10 and 18.

Basically we adopt existing techniques for continuously monitoring the results of the traditional queries. A straightforward way is to continuously monitor the traditional query result and report once the update occurs. However, since most state-of-art techniques for continuous monitoring queries compute and output their results incrementally, it is seamless to report updates based on these online algorithms. For instance, the techniques in the papers [3, 4] can work as a traditional query module to find the loyal objects within the query range or the influence zone for a majority of the recent time. As this part of work has already been done and our aim is to support a variety of traditional queries in our loyalty query framework, in this paper we focus on efficiently processing of the loyalty queries.

Loyalty query module. If we assume the attributes of an object is varying continuously such as a moving object, the number of updates during a time period is finite. For a specified loyalty query, it receives updates from the traditional query module in the form of an update stream $U = \{u_1, u_2, u_3, ..., u_n\}$. The updates arrive in the time order. We process the updates continuously in the loyalty query module and output the results to users.

Our query algorithm is triggered only when the update arrives or a possible result change of the loyalty query happens. Therefore, we can output updated results of the loyalty queries when the result changes. In other words, we report which object is newly added in the answer set or which object is removed from the set.

4. TOP-K LOYALTY QUERIES

4.1 Algorithm and Data Structures

Consider a top-1 loyalty query. If we draw the loyalty changes in a loyalty-time plane (see Figure 1), intuitively this problem is similar to finding the upper envelop in this plane. Similarly the top-k query is to retrieve kth upper envelops. This problem can be solved by the line sweep algorithm in computational geometry. Given N line segments in the plane, the Bentley-Ottmann sweep algorithm [2] maintains the exact vertical ordering of the intersections of the line segments, when the vertical line sweeps the plane from left to right. The total cost is $O((N+M)\log N)$ where M is the number of intersections of the line segments. In the worst case, the number of intersections M can be $O(N^2)$ and the overall complexity can be $O(N^2\log(N))$. In our problem, we use N to denote the number of updates issued in the last T time units. Then, the amortized cost of the Bentley-Ottmann algorithm is $O(N\log N)$ for each update. In this paper we present an algorithm to answer the top-k loyalty queries in $O(\log N)$ time for each update. The space requirement of our algorithm is $O(N)$.

Before we describe the algorithm, we show the observation for handling updates to enable the efficient computation.

States of objects. The state of an object o denotes whether the loyalty of o is increasing, stationary or decreasing. The state of o can be derived by computing the derivative of $loyalty(o,t)$. Therefore, $state(o,t) = \frac{d}{dt}loyalty(o,t) = Q(o,t) - Q(o,t-T)$ We can see that $state(o,t)$ depends on the traditional query result at the current time $Q(o,t)$ and the result T time before the current time $Q(o,t-T)$. Moreover, there are only three types of states: increasing (1), stationary (0) and decreasing (-1).

Echo updates. As soon as an *original* update arrives from the traditional query module, we know the traditional query result at the current time $Q(o,t)$ changes. Moreover, these updates will expire from the sliding window after T time, which will affect $Q(o,t-T)$. Therefore, we clone a series of *original* updates and make them take effect after T time. These updates are annotated as *echo* updates. For example, in Figure 1, we retrieve an original update at time 5. Given $T=10$, the echo update is created at time 15. The updates stand for both original and echo updates in the rest of the paper, unless mentioned otherwise.

Determining states. When we receive an original update u from a traditional query, we update the current query result $Q(o,t)$. Then we create an echo update u'. The timestamp of u' is $t+T$ and we also attach the new query result. Therefore, $state(o,t)$ can be computed by maintaining $Q(o,t)$ and $Q(o,t-T)$ correctly.

Data structures. In order to efficiently maintain the top-k loyal objects over sliding windows and the sequence of future updates and events, our algorithm maintains the following data structures:

Update queue U (a FIFO data structure) is utilized to maintain a sequence of echo updates. Each update is associated with the timestamp t_4 when it will be issued, the object o and the updated traditional query result $Q(o,t)$. The echo updates are created in the sequence of the original updates. Therefore, we can simply use a FIFO to organize the echo updates.

Border object BO is denoted as the $(k+1)$th loyal object at time t. We set BO empty if the number of objects is less than $k+1$. We define the *border line* indicating the $(k+1)$th line segment which divides the top-k lines and the remaining lines in the loyalty-time plane. In our algorithm only the line intersections related to the border line are processed. Consider a more complicated example of the top-2 loyalty query in Figure 2. The object o_3 is the 3rd loyal object from t_1 to t_3. Hence, $BO = o_3$ from t_1 to t_3. We mark the border line with a bold polyline in Figure 2.

Event queue E (a priority queue) is utilized to maintain a sequence of potential future events. *Events* denote the potential future result changes of the loyalty queries. The result changes occur only when the border object swap its order of the loyalty with another object. In the loyalty-time plane, the event is created when one line will potentially intersect the border line in the future. If an event is created at time t, each event is associated with the *signatures* of the border object BO and another o at time t. A signature is the identification of the last update of an object. The signature of o will be changed if any update or event related to o is processed. An event is invalid and will not be processed if the signature of BO or o of the event is not up-to-date. The event is inserted into the event queue with the timestamp t' where t' is the potential intersecting time. Consider the example in Figure 2. We can predict that the

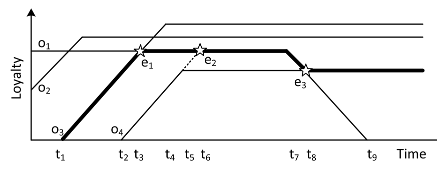

Figure 2: Example of Top-2 Loyalty Queries

line segment of o_1 will potentially intersect the border line at t_3. Therefore, an event is created to handle the intersection.

Top-k sets $A = A_+ \cup A_= \cup A_-$ maintain the objects geometrically *above* the border object, namely the top-k loyal objects. A is divided into three subsets according to the states of the objects. A_+, $A_=$ and A_- are the subsets of the top-k objects with increasing, stationary and decreasing states respectively. Each subsets is organized in a binary search tree and the elements in the subset are sorted in the decreasing order of their loyalties. Consider the example of Figure 2. A contains two objects o_1 and o_2 at t_1. $A_+ = \{o_2\}$ and $A_= = \{o_1\}$.

Bottom sets $B = B_+ \cup B_= \cup B_-$ maintain the remaining objects *below* the border object. B is also divided into B_+, $B_=$ and B_- according to the states. Note that unlike other subsets, B_- can be organized just in a list without sorting their loyalties. B_+ and $B_=$ are represented explicitly in the binary search trees with the decreasing order of loyalties. Consider the example of Figure 2. B contains one object o_4 at t_4 and $B_+ = \{o_4\}$.

Solution overview. Before we present the details of our algorithm for processing top-k loyalty queries, we show the main idea of our algorithm. The algorithm uses a sweep line approach to process updates and create events for handling the possible result changes. The algorithm is triggered when 1) an original update arrives from traditional query module, or 2) an echo update arrives from the update queue, or 3) an event arrives from the event queue. We make sure that our algorithm correctly maintains the border object and the objects in top-k set. An event is created if a possible result change of the loyalty query will occur in the future.

Algorithm 1 ProcessUpdate(u)

```
 1: Determine $state(o,t)$ and update $loyalty(o,t)$.
 2: if $o \in A$ then /* $o$ is in the top-$k$ set */
 3:     Remove $o$ from the subset $A_i$
 4:     Add $o$ into the corresponding subset $A_j$
 5: else if $o \in B$ then /* $o$ is in the bottom set */
 6:     Remove $o$ from the subset $B_i$
 7:     Add $o$ into the corresponding subset $B_j$
 8: else if $o \notin BO$ then /* $o$ is a new comer */
 9:     if $|A| < k$ then /* # of objects less than $k$ */
10:         Add $o$ into $A_+$
11:     else if $BO = \emptyset$ then /* # of objects is $k$ */
12:         $BO = o$
13:     else
14:         Add $o$ into $B_+$
15: HandleSetVariation($BO$, $A$, $B$)
16: Update the signature of $o$.
```

Processing updates. When a new update arrives from the update queue, we first recompute the state and loyalty of the corresponding object. Then, the object is moved to the correct subset. As the position of the object in a subset may be changed, we check the subsets and the border object and create the possible events.

Algorithm 1 shows our algorithm for processing a newly arriving update. First we determine the state of the object o based on the query results on both slides of the sliding window (see line 1). If the update is original, we create an echo update by cloning the original update and insert it into the update queue P. Since the state of the object o changes, we move o into the corresponding subset based on its state and current position(lines 2–8). As the orders of elements in the subsets may change, we check the variation related to the border line and create new events to handle the future intersection(line 15). Finally we update the signature of o (line 16) as the state of o changes.

Handling set variations. We observe that any intersection related to the border line is associated with the line segment immediately above or below the border line in each subset. Another important observation is that the two line segments with the same state(in the same subset) will not intersect each other. Therefore, we only check the last elements(objects with the minimal loyalty) in $A_=$ and A_-, and the first elements(objects with the maximal loyalty) in B_+ and $B_=$, which are the only potential line segments (objects) to first intersect the border line without considering the new updates in the future In Figure 2, $A_=$ contains two objects o_2 and o_3 at t_6, and the last element in $A_=$ is o_2. Then, we consider the state change of the border object BO. Based on the state of the border line, two events are created to handle the possible intersections.

Creating events. To create a new event, we first compute the intersecting time t' of the two lines segments. Then, the event e is created and inserted into the event queue E with t'. Note that it is important for us to store the signature information of o and BO with event e. The change of the signature of o indicates that the state or position of the object has been updated before the event occurs. Therefore, the event is invalid and will not be processed.

Processing events. When an event e arrives from the event queue E, we first check the validity of the line intersection by verifying the signatures. If it is valid, we swap the positions of BO and o. Since the subsets and BO are changed, we again check the set variations and create the possible intersections.

Handling objects with zero or maximum loyalty. Note that in the above algorithms we do not especially handle the objects with zero loyalties or maximum loyalties(the loyalty is T). Here, we show that these objects can be processed more efficiently. For an object o with $loyalty(o, t) = 0$ and $state(o, t) = 0$, we simply remove o from the subsets. For the objects with $loyalty(o, t) = T$ and $state(o, t) = 0$, we maintain a list F to store the objects instead of placing them in $A_=$. We can save the cost because maintaining the list is constant in time.

Example 1: Consider the top-2 loyalty query shown in Figure 2. Initially, there are two objects o_1 and o_2 with non-zero loyalties. An update of o_3 arrives at t_1. o_3 becomes the border line object(line 11 in Algorithm 1). We mark the border line with a bold line in Figure 2. Then, we check the set variation. Since BO has been changed, we create an event e_1 with the last element in $A_=$ (o_1) for possible order swapping at t_3. We mark the created event with a star. Note that we only create and process the events (intersections) related to BO. An update of o_4 arrives at t_2. We check subsets variation and no event is created. Event e_1 is processed at t_3. o_3 is moved into A_+ and o_1 becomes the border object. We check the set variation and create an event e_2 with o_4 at t_6. After that o_4 issues another update at t_5 and the signature of o_4 is changed. Therefore, e_2 is invalid and is not processed at t_6. At t_7, o_4 issues an update and the state of BO is changed. After checking the variation, e_3 is created similarly.

4.2 Analysis

Proof of Correctness. In the proposed algorithm, we make the border object BO present the $(k + 1)$th loyal object correctly. All the potential events (intersections) related to BO are created and processed. Therefore, we always make $\max_{o \in B} \{loyalty(o, t)\} \leq loyalty(BO, t) \leq \min_{o \in A} \{loyalty(o, t)\}$ and $|A| \leq k$. In our algorithm the objects in top-k set cannot be changed unless BO is changed. As a consequence, our algorithm correctly determines the top-k loyal objects.

Query performance analysis. We first analyze the time complexity of our algorithm. As we use binary search trees to maintain the subsets, the cost of inserting or removing an object in a subset of A is $O(\log k)$ and the corresponding cost in a subset of B is $O(\log L)$ where L is the number of objects which have updates in the last T time unit. The cost of insertion in the update queue is $O(1)$ because the update queue is a FIFO. Let M be the number of events processed in the last T time units and M' be the number of events created in the last T time units. Note that some created events may become invalid and will not be processed in the future. As the event queue is organized by a priority queue, the cost of insertion in the event queue is $O(\log M')$, where M' is also the size of the event queue. Let N be the number of updates issued in the last T time units. For each processed update and event, the algorithm creates constant number of events. Therefore, $M' = O(N + M)$. Then, the total cost in the last T time units is $O((N + M)(\log M' + \log k + \log L) = O((N + M)(\log (N + M) + \log k + \log L)$. Note that $k \leq L$ and L is usually much smaller than the total number of objects n. Therefore, the total cost in the last T time units is $O((N + M)(\log (N + M) + \log L)$. In Theorem 1 we prove that the number of processed events is at most twice of the number of updates, i.e., $M \leq 2N$. The theorem is non-trivial, because two other events will be created when processing the event.

THEOREM 1. *Given N updates, our algorithm processes at most* $2N$ *events.* $M \leq 2N$.

PROOF. Consider the loyalty-time plane and assume that each line segment presents an update in the plane (see Figure 2). The border line is actually one of the connected line segments that go through the plane from left to right. For an increasing line or decreasing line, it appears in the border line at most once, while a horizontal line may appear in the border line multiple times. The horizontal lines are only connected with the increasing and decreasing lines in the plane. Assume that the border line has at least two line segments. One horizontal line on the border line must connect with one increasing line or decreasing line. Let P be the number of line segments on the border line and Q be the number of increasing and decreasing lines. In the worst case, every horizontal line segment is associated with one increasing or decreasing line. Therefore, $P \leq 2Q$. Each connected vertex on the border line presents a processed event. Consequently, we prove that $M \leq 2N$. □

Theorem 1 indicates that the number of processed events is at most twice of the number of updates. We can derive that $M = O(N)$. Moreover, $L \leq N$ because the number of objects which have updates will not larger than the number of updates. Therefore, the total cost of our algorithm in the last T time units is $O(N(\log N))$. The cost for each update is $O(\log N)$.

Proof of Optimality. We prove that our algorithm is optimal in the worst case by showing that the complexity of our algorithm meets the lower bound cost of the problem [9].

Space analysis Next, we investigate the space requirement of our algorithm. The space of the update queue is $O(N)$ where N is the number of updates issued in the last T time units. The size of the event queue is $O(M')$. According to the above analysis, $M' = O(N)$. The size of each subset is $O(L)$. If we do not consider the objects with maximum loyalties, then $L \leq N$. Therefore, for each top-k loyalty query, our algorithm uses $O(N)$ space.

4.3 Pruning

Although the algorithm is already optimal for solving the top-k loyalty queries in terms of time complexity, in this subsection we show that we can further prune some of the updates from the computation of the final results according to the following theorem.

THEOREM 2. *Let* o_k *be the object with the minimal loyalty in* A *and* o *be any object in* O. o *will not be a result of top-*k *loyalty query in the next* $(loyalty(o_k, t) - loyalty(o, t)/2$ *time, where* t *is the current timestamp.*

PROOF. Consider that o_k becomes decreasing and o becomes increasing at t. Let $d = (loyalty(o_k, t) - loyalty(o, t))/2$. o will be always below o_k in the time period $[t, t+d)$. Thus, $loyalty(o_k, t+\Delta t) > loyalty(o, t+\Delta t)$ where $0 \leq \Delta t < d$. Consequently, we prove Theorem 2. $\Box$

Based on the theorem, it is not difficult to ignore the update computation of $o \in O$ in time period $[t, t+d)$. The detail of the algorithm can be found in our technical report [9].

5. RELATED WORK

The problem is related to the kth upper envelope (also known as k-level arrangement) problem [2] in computational geometry. However, the current techniques [6, 1, 8] can only solve the problem for $k = 1$. In database community, processing aggregate queries over data stream [7, 12, 10] has been extensively studied. The difference is that it can only solve the problem by sampling in the discrete time domain, which limits the precision and efficiency. Also, the existing work of continuously processing the spatial-temporal queries [5, 3, 4, 11] can be used as the traditional queries module in our framework.

6. EXPERIMENTS

In the experiments, we focus on evaluating the performance of the proposed algorithm for answering top-k loyalty queries. Therefore, we do not count the cost of computing traditional query results and assume that all the inputs are in the form of object updates. Synthetic data is generated by a two state Markov chain model, which has many applications as statistical models of real-world processes. For each object o_i,

$Pr(Q(o_i, t+1) = 1 | Q(o_i, t) = 0) = p_i$
$Pr(Q(o_i, t+1) = 0 | Q(o_i, t) = 0) = 1 - p_i$
$Pr(Q(o_i, t+1) = 0 | Q(o_i, t) = 1) = p'_i$
$Pr(Q(o_i, t+1) = 1 | Q(o_i, t) = 1) = 1 - p'_i$

p_i and p'_i are uniformly chosen from $[0, m]$ for each object. The data set consists of 10 million random updates with n objects.

Table 1: Experimental parameters. (Default values in bold)

Parameter	Range
Sliding window size T (× 1000)	**10**, 25, 50, 75, 100
# of objects n (× 1000)	**1**, 5, 10, 15, 20
# of results k	1, 10, 20, 50, **100**, 150, 200
Probability parameter m	0.0001, **0.001**, 0.01, 0.1, 1

To the best of our knowledge, we are the first to study the problem of top-k loyalty queries. Therefore, we use the Bentley-Ottmann algorithm as our competitor called *BO* below. Our base loyalty query processing algorithm is called *LQ*. The loyalty query processing algorithm optimized by using the pruning rule is called *LQPR*. Note that all the figures are in the logarithmic scale.

In Figure 3, we perform experiments on syntectic data sets to conduct a more detailed evaluation. We study the effect of varying k and T in Figure 3(a) and Figure 3(b). The similar tendency can be observed on the synthetic data set. Figure 3(a) shows that the pruning rule does not work well when the sliding window size T is small. The reason is that the number of updates generated with certain probability in a small sliding window is small. Therefore, not many updates can be pruned according to the pruning rule.

In Figure 3(c) and Figure 3(d), we vary the number of objects n and the probability m used in generated synthetic data and study the effect on the algorithms. Figure 3(c) shows that the processing time of our algorithms increases with increase in n. This is because the number of objects which have updates in the sliding window L increases with larger n. Figure 3(d) shows that the performance of our algorithms remains unaffected with increase in the frequency of updates, although we vary m in a very large scale. LQPR does not show a good pruning power when $m = 0.0001$ because the number of updates in the sliding window is too small so that few updates can be pruned.

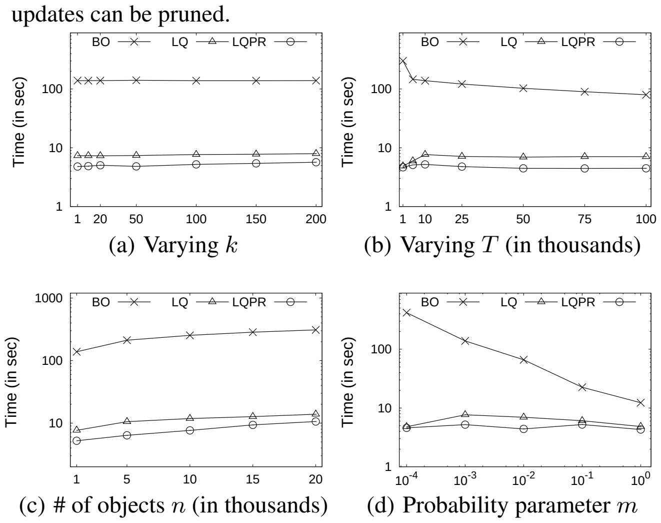

Figure 3: Performance evaluation on the synthetic data

7. CONCLUSION

We introduce the loyalty queries for a variety of applications. We present efficient algorithms to answer the top-k loyalty queries. We prove the lower bound cost of the problem and present a detailed complexity analysis to show that our algorithm is optimal. We verify this by an experimental evaluation and demonstrate the efficiency of our approach.

Acknowledgments: The third author is partially supported by ARC DP0987557, ARC DP110102937, ARC DP120104168 and NSFC61021004.

8. REFERENCES

[1] Basch, J., Guibas, L.J., Hershberger, J.: Data structures for mobile data. J. Algorithms 31(1), 1–28 (1999)
[2] Bentley, J.L., Ottmann, T.: Algorithms for reporting and counting geometric intersections. IEEE Trans. Computers 28(9) (1979)
[3] Cheema, M.A., Brankovic, L., Lin, X., Zhang, W., Wang, W.: Multi-guarded safe zone: An effective technique to monitor moving circular range queries. In: ICDE. pp. 189–200 (2010)
[4] Cheema, M.A., Lin, X., Zhang, W., Zhang, Y.: Influence zone: Efficiently processing reverse k nearest neighbors queries. In: ICDE. pp. 577–588 (2011)
[5] Gedik, B., Wu, K.L., Yu, P.S., Liu, L.: Processing moving queries over moving objects using motion-adaptive indexes. IEEE Trans. Knowl. Data Eng. 18(5), 651–668 (2006)
[6] Hershberger, J.: Finding the upper envelope of n line segments in o(n log n) time. Inf. Process. Lett. 33(4), 169–174 (1989)
[7] Li, J., Maier, D., Tufte, K., Papadimos, V., Tucker, P.A.: Semantics and evaluation techniques for window aggregates in data streams. In: SIGMOD Conference. pp. 311–322 (2005)
[8] Russel, D., Karavelas, M.I., Guibas, L.J., Guibas, L.J.: A package for exact kinetic data structures and sweepline algorithms. (2007)
[9] Shen, Z., Cheema, M.A., Lin, X.: Loyalty-based retrieval of objects that satisfy criteria persistently. In: UNSW Technical Report, 2012. ftp://ftp.cse.unsw.edu.au/pub/doc/papers/UNSW/201224.pdf
[10] Wang, S., Rundensteiner, E.A., Ganguly, S., Bhatnagar, S., Bhatnagar, S.: State-slice: New paradigm of multi-query optimization of window-based stream queries. In: VLDB. pp. 619–630 (2006)
[11] Xiong, X., Mokbel, M.F., Aref, W.G.: Sea-cnn: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In: ICDE. pp. 643–654 (2005)
[12] Zhang, R., Koudas, N., Ooi, B.C., Srivastava, D.: Multiple aggregations over data streams. In: SIGMOD Conference. pp. 299–310 (2005)